TENTAMEN

Objektorienterad programmering

- Uppgifterna är inte ordnade efter svårighetsgrad.
- Programkod som finns i tentamenstesen behöver ej upprepas.
- Det räcker med enbart relevanta kodavsnitt, övrig kod ersätts med ... (aldrig `import`, hjälpklasser såsom `Main`, `main`-metod som bara anropar relevant metod, etc. om det inte tydligt framgår i uppgiften att det förväntas).
- Oprecisa eller alltför generella (vaga) svar ger inga poäng. Konkretisera och/eller ge exempel. Det är aldrig någon risk att vara övertydlig!
- Programkod skall skrivas i Java 5, eller senare version, och vara indenterad och renskriven.
- Onödigt komplicerade lösningar kan ge poängavdrag.
- Givna deklarationer, parameterlistor etc. får inte ändras om det inte uttryckligen är en del av uppgiften. Fråga i oklara fall!
- I en uppgift som består av flera delar får du använda dig av funktioner, klasser, etc. från tidigare deluppgifter, även om du inte löst dessa.
- Läs igenom tentamenstesen och förbered eventuella frågor. Pelle kommer att besöka tentamenssalen 9:30 och 11:00.
- Kom ihåg att inte fastna på en uppgift. Bestäm i förväg din egen tidsgräns per uppgift.

Betygsgränser:
- 3: 24 – 35 poäng
- 4: 36 – 47 poäng
- 5: 48 – 60 poäng

(Max 60 poäng.)

Lycka till!

Uppgift 1 Objektorientering allmänt (12p totalt)

(A) Förklara begreppen låg/hög koppling och kohesion. Beskriv så väl du kan med exempel varför kohesion och koppling i någon mån kan vara både individuellt höga/låga. (3p)

(B) Förklara kort hur ”mutation” kan utföras för icke-muterbara klasser (tänk String, BigInteger). (3p)

(C) Förklara skillnaden mellan specifikation och implementation. (3p)

(D) Förklara skillnaden mellan klass- och instansvariabler. (3p)

Uppgift 2 Beräkning med memorering (10p)

Collatz-, ”3n+1” eller ”Hagel”-problemet är ett öppet problem inom matematiken.
För alla värden vi prövat så leder funktionen nedan så småningom till att vi hamnar i cykeln {4, 2, 1, 4, ...} men det är inte bevisat att så är fallet för alla naturliga tal.
Collatzfunktionen kan i Java beskrivas enligt följande:

```
public class Collatz {
   private static final BigInteger THREE = BigInteger
      .valueOf(3);
   static BigInteger nextCollatz(BigInteger v) {
      if(!v.testBit(0)) { // Test if v is even.
         return v.divide(BigInteger.TWO);
      }
      return v.multiply(THREE).add(BigInteger.ONE);
   }
}
```

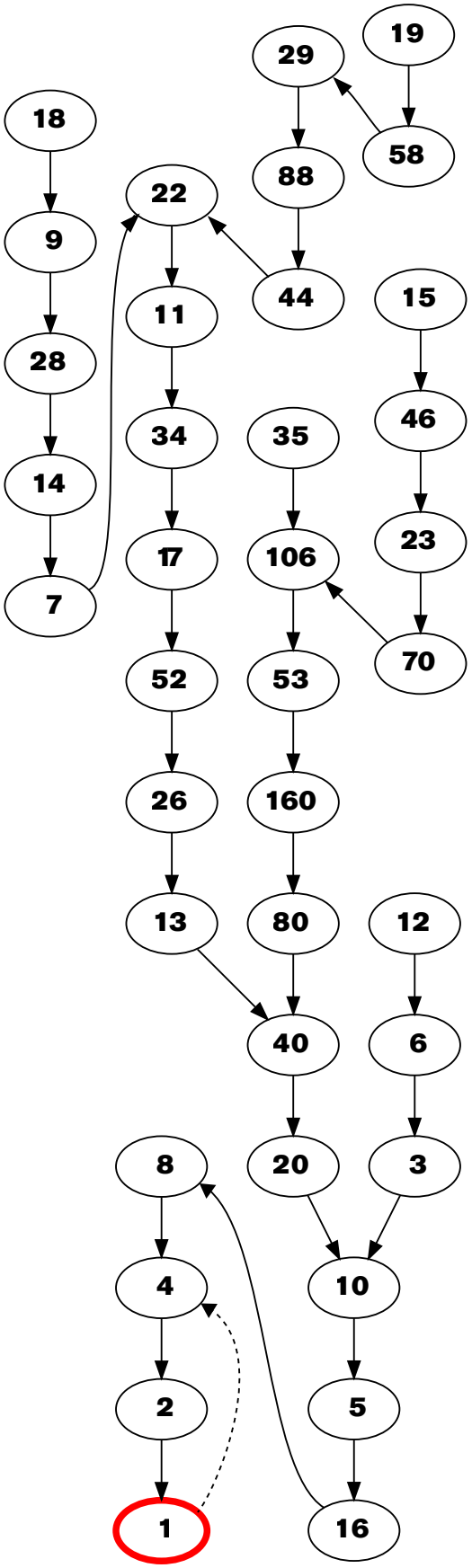
Funktionen returnernar `v / 2` om v är jämnt, annars `3 * v + 1`.
Vi ska beräkna medelvärdet för antalet iterationer funktionen tar på sig innan man når värdet 1. Definition av antal iterationer kan skrivas enligt följande:

```
static int getCollatzIterations(BigInteger v) {
   int count = 0;
   while(!v.equals(BigInteger.ONE)) {
      count++;
      v = nextCollatz(v);
   }
   return count;
}
```

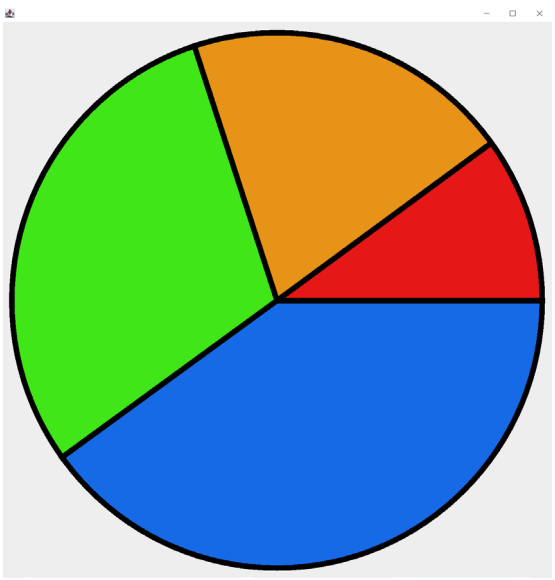
Genom att direkt anropa `getCollatzIterations` i en loop för talen {1, ..., 10^7} så använder min dator 41.5 sekunder innan den är klar. Här kan vi dock göra något slugt då vi inser att vi inte behöver följa en iteration till slutet om vi redan känner till antalet steg det tog att komma till ett visst tal förra gången.
Din uppgift blir att ***med hjälp av en Map*** ”minnas” värden där vi redan vet hur många iterationer man är från värdet 1 – som en konsekvens kan vi då bryta vår iteration tidigt i många fall.
Min egen implementation gick från 41.5 till 18.7 sekunder för 10 miljoner tal, vilket är signifikant.

Tips: Kör din implementation i huvudet och kolla att den verkar stämma för 1, 2 och 3.
Tips: Det kan vara klokt att implementera den naiva versionen först...

Grafen till höger visar hur funktionen ser ut för iteration för talen 1 till och med 20.



Uppgift 3 Tårtdiagram (15p totalt)



Din uppgift är att implementera en klass för att rita (enkla) tårtdiagram.
Klassen ska vara en (utökning av en) JPanel. Följande bit programkod ska ge resultat enligt illustrationen till höger:

```
public static void main(String[] args) {
   final JFrame f = new JFrame();
   final JPanel p = new PiePanel(new double[] {
      1, 2.0, 3, 4 });
   p.setPreferredSize(new Dimension(400, 400));
   f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   f.add(p);
   f.pack();
   f.setVisible(true);
}
```

Saker du inte behöver tänka på:
Linjegrovlek runt diagrammet och mellan tårtbitar (men linjer bör finnas med).
Exakt vilka färger som används men två ***intillliggande tårtbitar får inte ha samma färg.*** (I illustrationen har jag använt mig av

```
g.setColor(Color.getHSBColor((float) (angle / 360.0),
   0.9f, 0.9f));
```

Eventuella marginaler/avstånd till komponentens kanter (men hela diagrammet ska alltid rymmas inom komponenten).
Diagrammet ska förstås ritas om och skalas beroende på komponentstorlek.

Obs: du får anta att inputarrayen till konstruktorn varken är `null`, tom eller innehåller negativa värden.

Tips: det är lämpligt att använda sig av
`Graphics.fillArc(x, y, width, height, startAngle, arcAngle).`
för att fylla tårtbitarna. `startAngle = 0` ger vinkel ”klockan 3” eller ”österut”.
Notera att samtliga parametrar till `fillArc` är av typ `int`.

Tips: cirkelns periferilinje ritas enklas ut som m.h.a. drawOval(x, y, width, height).

Tips: Linjerna bör ritas ut efter tårtbitarna där koordinat på circkelns periferi fås av

```
x = Math.cos(theta) * radiusX + centreX;
y = -Math.sin(theta) * radiusY + centreY;
```

Uppgift 4 Memoryspel (23p totalt)

Spelbräde där användaren identifierat två par (0 och 4) och valt första kortet från nästa par (7).

Din uppgift blir att skriva en modell och controller (som sammanfaller med view) för ett enkelt memory-spel.
Spelet går ut på att användaren på så få drag som möjligt ska identifiera alla par. ”Korten” styrs genom JButtons där texten är satt till tom sträng när de ligger nedåtvända.
Då användaren klickar på ett första ”kort” visas siffran (som modellen lagrar).
Vid klick på ytterligare ett kort visas siffran på det andra kortet också.
Om talen överensstämmer så ska modellen se till att poängen uppdateras. Controllern ska ansvara för att knapparna blir markerade gröna (`button.setBackground(Color.green)`) samt att de inaktiveras (`button.setEnabled(false)`) då ett par identiferats.
Texten som visar poängen uppdateras löpande – oavsett om användaren hittat ett par eller ej.

Saker som inte är viktiga:
Typsnitt för knapparna.
Att Score skrivs ut med ett begränsat antal decimaler.

Saker som är viktiga:
Knapparna ska ligga i ett rutnät som har bredd och höjd enligt modellen.
Score och Restart-knapp ska finnas längst ned.

Modellen görs enklast enligt följande specifikation:

```
public class MemoryModel {
   // instansvariabler ...

   // Sets up a board with size width x height.
   // You may assume that width * height always
   // is even.
   public MemoryModel(int width, int height) {...}

   // Returns the score computed as the number of pairs found divided by the number of
   // (pairwise) guesses.
   public double getScore() { ... }

   // Returns the height of the board.
   public int getHeight() { ... }

   // Returns the width of the board.
   public int getWidth() {
      return this.width;
   }
```

```
// Returns true if all pairs have been found.
public final boolean isGameOver() { ... }

// Returns the value at coordinate <x, y>.
public int getValue(int x, int y) { ... }

// Makes a guess, returns true if the values at {<x1, y1>, <x2, y2>} are identical.
// Should update the number of guesses as well as the score.
public boolean guess(int x1, int y1, int x2, int y2) { ... }

// Restarts the game:
// Resets number of guesses and score.
// Shuffles the cards. This can most conveniently be done by using a list for
// all card values and then call Collections.shuffle(cardList) to shuffle
// them pseudo-randomly.
public void restart() { ... }
```

Tips: Använd en kombination av BorderLayout (för övergripande struktur) och bygg upp rutnätet med knapparna via GridLayout. Du behöver högst sannolikt använda flera olika paneler.

Exempel på main-metod för att starta programmet (du behöver inte skriva någon `main`, controller- och modellklass räcker):

```
public static void main(String[] args) {
   final JFrame f = new JFrame("Memory!");
   // Creates a new MemoryModel, 4 columns wide, 6 rows tall,
   // and passes it to a new controller.
   final JPanel p = new MemoryController(new MemoryModel(4, 6));
   f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   f.add(p);
   f.pack();
   f.setVisible(true);
}
```

Utdrag ur Javas API. Obs: Man behöver inte använda alla dessa klasser. Man får också använda annat från Javas API.

```
Class AbstractButton
   void setText(String text)
```
Sets the button's text.

```
Interface ActionListener
   void actionPerformed(ActionEvent e)
```
Invoked when an action occurs.

```
Class Component extends Object
   void repaint()
```
Repaints this component.

```
Class HashMap<K,V> extends AbstractMap<K,V> extends Object implements Map<K,V>
   HashMap()
```
Constructs an empty `HashMap` with the default initial capacity (16) and the default load factor (0.75).

```
   V get(Object key)
```
Returns the value to which `key` is mapped, or `null` if this map contains no mapping for the key.

```
   V put(K key, V value)
```
Associates the specified value with the specified key in this map.

```
Class IllegalArgumentException extends Exception
   IllegalArgumentException()
```
Creates a new `IllegalArgumentException`.

```
Interface Iterator<E>
   boolean hasNext()
```
Returns true if the iteration has more elements.

```
   E next()
```
Returns the next element in the iteration.

```
Class JButton extends AbstractButton extends … extends Object
   JButton(String text)
```
Creates a button with text.

```
Class JComponent extends Container extends Component extends Object
   int getHeight()
```
Returns the current height of this component.

```
   int getWidth()
```
Returns the current width of this component.

```
   void paintComponent(Graphics g)
```
Calls the UI delegate's paint method, if the UI delegate is non-null.

```
Class JFrame extends Frame extends Container extends Component extends Object
   JFrame(String title)
```
Creates a new, initially invisible Frame with the specified title.

```
   void setLayout(LayoutManager manager)
```
Sets the LayoutManager.

```
Class JPanel extends JComponent extends Container extends Component extends Object
   JPanel()
```
Creates a new JPanel with a double buffer and a flow layout.

(fortsättning på nästa sida)

```
Interface List<E>
   boolean add(E e)
```
Appends the specified element to the end of this list.
```
   Iterator<E> iterator()
```
Returns an iterator over the elements in this list in proper sequence.
```
   E get(int index)
```
Returns the element at the specified position in this list.
```
   int size()
```
Returns the number of elements in this list.

```
Class LinkedList<E> extends AbstractCollection<E> extends Object implements List<E>
   LinkedList()
```
Constructs an empty list.

```
Class Object
   boolean equals(Object obj)
```
Indicates whether some other object is ”equal to” this one.